Compiler Construction (Lab-1)

15/01/2023

~~Cls-1~~

- Compiler – TOC ہی اگلا part
- Regular expression → Flex/Bison
  (+, *) — Software
- Compiler → Executes whole code
- Interpreter → Execute code line by line

→ low level (Assembly, ML) — binary
→ high level (c, Java, ... etc)
→ compiler :-
c → Gcc, c++ → g++,
Java → JDK

(e) High level code
↓
Compiler
↓
Assembly
↓
ML
Compiler

(Java) . Java
↓ ↓
~~compiler~~ Compile
↓
Bytecode (Intermediate Language)
↓
JVM (Java Virtual machine)
↓
ML
Hybrid compiler

- Faster Execution → Compiler.
- Faster Error Detection → Interpreter.

Ex:- c (compiler), python (Interpreter), Java (compiler + Interpreter)

Q. diffrenec between compiler & interpreter.

Q. '2 + 2'
Integer — operand — Integer

cls-1

Book:- Compilers Principles, Techniques & Tools — Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D.

* Prospect :- NLP
  ↳ Machine Translation

Hybrid Compiler:- p-3

- Java
  ↓
  compiler (JDK(v))
  ↓
  Byte code
  ↓
  JVM
  ↓
  ML

Source code
↓
[Translator]
↓
Intermediate Program → [Virtual Machine] → output
input → [Virtual Machine]

* Language Processing System:- p-4

Execution Time
Compile Time Run T

Source Program
↓
[Preprocessor]
↓
Modified source program
↓
[compiler]
↓
target Assembly Program
↓
[Assembler]
↓
relocatable machine code (RAM)
↓
[Linker/Loader]
↓
Target Machine code

Structure of a compiler :- P-4

Analysis Part/ Frontend	Synthesis Part / backend
→ breaks up the source code into constituent pieces & impose a grammatical structure. (rules, reg, terms CFG)	→ Constructs the desired target program from the intermediate representation & the information in the symbol table. (assembly)
→ Using the grammatical structure to create an intermediate representation (three Address code / Four AC) of the source program.	
→ If the part detects that source program is either syntatically ill formed or semantically unsound, then (heart of compiler) It must provide information messages, so the user can take the corrective action.	
→ The analysis part also collects information about source program & stores in a data structure called a symbol table, which is passed along with the interpresentation to the synthesis part.	

CC (T-2) 19/4/23

Compiler Phase (P-7)

↓ Position = initial + rate * 60

input from source code receive or output from token or

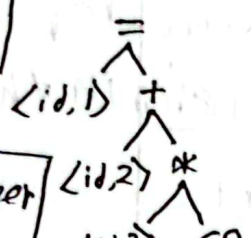
Lexical Analyzer

↓ <id,1><=><id,2><+><id,3><*><60>

identifier, attribute

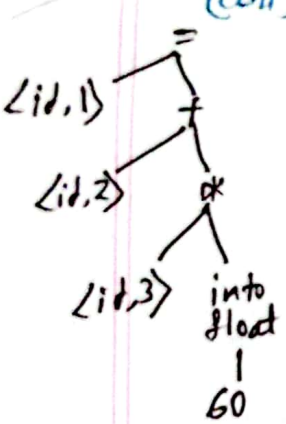
input token & output tree. (Parser) (error detect)

Syntax Analyzer



(CGF)

Semantic Analyzer



Intermediate Code Generator

t1 = intofloat (60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Code Optimizer

t1 = id3 * 60.0
id1 = id2 + t1

assembly

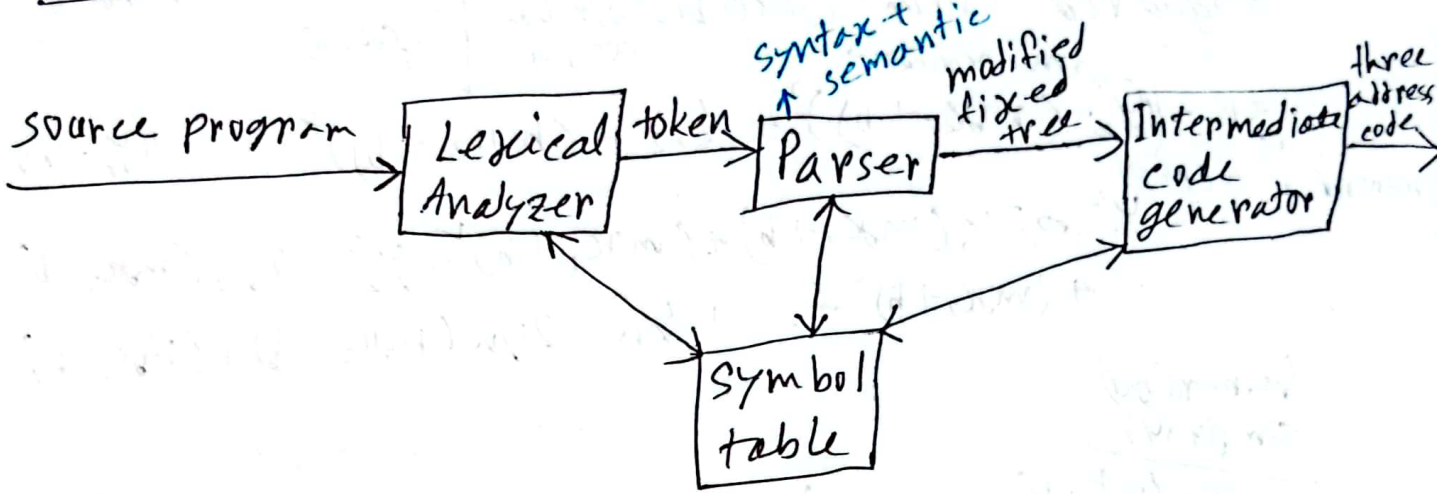
Code Generator

LDF R2, id2
MULF R2, R2 #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Symbol table

1	position	--
2	initial	--
3	rate	--

Model of compiler fronted using the phases:



CCL-2

22/1/23

Task-1: take a string as input & display weather they are integer or operator.

input :- 2+2 | output :- integer operator integer
5*8 | integer operator integer

Task-2:- from text file

input :- int x = 10; float f; double d = 20;

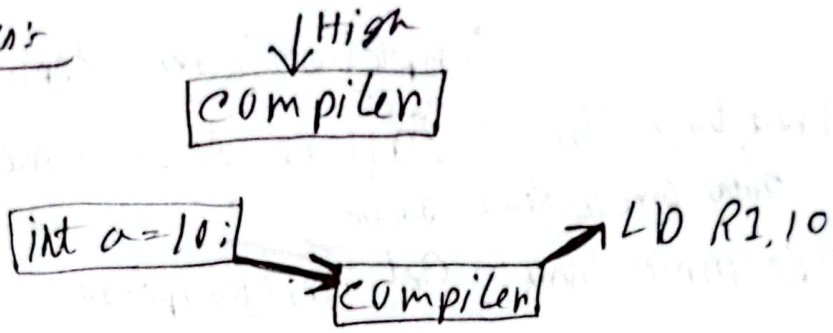
output :- datatype variable operator integer
datatype variable
datatype variable operator float

CC (T-3)

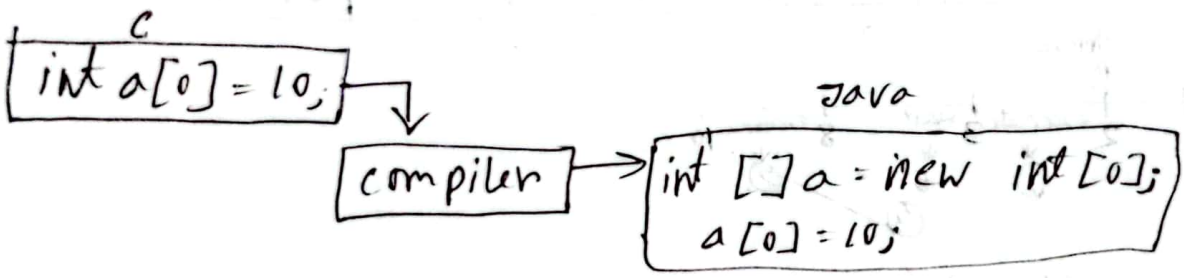
25/1/23

* compiler এর কাজ :- input → High level language
output → low level language } translate

* How does compiler works:



Source to Source Compiler:- এর high level language কে অন্য এক high level language এ convert.

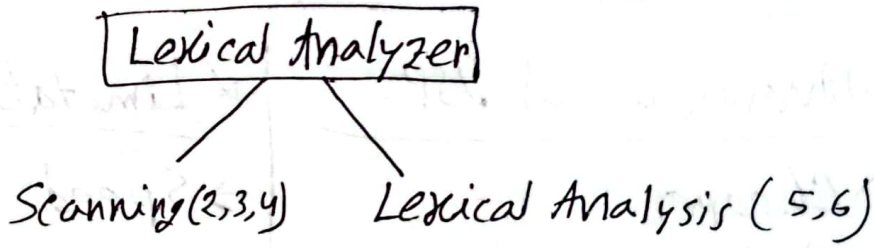


Q. A compiler that translet a high ~~level~~ level language into another high level language is called Source-to-source compiler / translator.

What advantages are there to using "C" as a target language for a compiler?

Lexical Analyzer:

1) First phase of the compiler
2) Stripping ~~out~~ out comments & whitespaces (blank, newline, tab & perhaps other characters that are used to separate tokens in the input)
3) Read the input characters of the source program, group them into lexemes.
4) Interact/construct the symbol table.
   operator
5) Produce as output a sequence of tokens from each lexeme in the source program.
6) The system of tokens is sent to the parser of syntax analysis.



CC (T-4) 1/2/25

Lexical Analyzer

Scanning
→ char scan
রাখ unnecessary
space & etc
remove করা

Lexical Analysis
→ lexeme
→ Pattern (set of Rules Reg.)
→ token (lexem যদি pattern satisfy করে or token)

* Lexeme:-

→ A lexeme is a sequence of characters in the source program.

→ It matches the pattern for a token & is identified by the lexical analyzer as an instance of that token.

$x = a + b * 2;$ → lexemes are :- { x, =, a, +, b, *, 2, ; }

* Token:- <token-name, attribute-value>

symbol table এর position & mandatory না

→ It is a pair consisting of token name & optional attribute value

→ The token name is an abstract symbol to represent a lexical unit. For example a particular keyword, or a sequence of input characters denoting an identifier.

→ We will offter refer to a token by its token name.

$x = a + b * 2;$

symbol table →

1	x	-
2	a	-
3	b	-

Total token = 8

<id, 1>	<x>
<=>	
<id, 2>	<a>
<+>	
<id, 3>	<b>
<*>	
<=>	
<;>	

<u>Symbol table:</u> It is a data structure. Containing a record for each <u>variable name</u> with for the attributes of the name. This data structure should be designed to allow the compiler to find the record for each name quickly & to store or retrive data from that record quickly.

<u>Example:</u> position = initial + rate * 60;

1	Position	int
2	initial	—
3	rate	—

<u>tokens are:-</u>

<id, 1> <=> <id, 2> <+>
<id, 3> <*> <60> <;> <int>

printf("Total = %d", x);

literal

<u>tokens:</u> <printf><(><"Total = %d"><,><x><)><;>

total Token = 7

"token = token" → literal

CC (T-5) 01/2/23

Pattern:

→ A pattern is a description (regular expression so arrow implement) of the form that the lexemes of a token may take.

→ In case of a keyword as a token, the pattern is just the sequence of characters that form the keyword

→ For identifiers & other token, the pattern is a more complex structure that is matched by many strings.

Example:

Token	Description	Sample Example
if	characters i, f	if
else	characters e, l, s, e	else
Comparison operator	< or > or <= or >= or == or !=	+, <, =
Identifier (id)	letter followed by letters & digits	pi, score, D2
number	any numeric constant	3.1415, 0, 6.0
literal	anything but ", surrounded by "'s	"core dumped"
Separator	any punctuation symbol. coma, semi colon	{}, (), []

Lexical Error:- Q. Lexical error detector or trap?

→ Lexical error is a sequence of characters that does not match the pattern of any token. Lexical phase error is found during the execution of the program.

→ Spelling error.

(Ex: eles, int 1x;
x=;

→ Exceeding length of an identifier or numeric constant
→ Appearance of ~~ille~~ illegal characters.

Ex:- find lexical errors:-

```
int main(){
    int p = 1 << 32;
    fi (f(x) = 24){
        print ("dragon");
        return 0;
    }
}
```

Panic Mode Recovery:- (Error recovery system)

→ This recovery technique may confuse the parser phase through lexical analyzer try to recover it until it can find a well-formed token at the beginning of what input is left.

Possible recovery ~~option~~:- options:-

→ Delete a character from the remaining input.
→ Insert a character from the remaining input.
→ Replace.
→ Transpose.

CC (T-6)

9/2/23

Lexical Analyzer

Pattern:

lexeme	pattern	token
(steem of character)	(Rules / Description) Designed by using Regular Expression (.l)	~~symbol attribute~~ <token name, attribute> ↓ symbol table.
int a=10; ↓ i,n,t,a,=,10,;		if

Grammer [Syntax Analyzer / Parser :- syntex error, x=a
[semantic

* Lexical Analyzer Error:-
  - → invalid identifier / variable declear, Ex: int 2x;
  - → Spelling mistake, Ex: int circle .... circle = 4
  - → datatype go limit cross.

Introduction to Syntax Analyzer / Parser:-

→ It is also known as parser.
→ It is generated a tree, namely parse tree.
→ It is very known phase for detecting the syntactical errors of the program.
→ It is introduced the Context Free Grammar (CFG) as the syntax of programming language constructs can be specified by CFG.
→ The parser obtains a string of tokens from the lexical analyzer.
→ Verifies that the string of token names can be generated by the grammer for the source language.

→ We expect the parser to report any syntex errors in an intelligible fashion & to recover from commonly occuring errors to ~~contai~~ continue processing the remainder of the program.

→ The parser constructs a parse tree & passes it to the rest of the compiler for further processing.

tree তৈরি করতে হবে, যেখানে root = operator, & child = variable

a+b →

token recive করতে হবে & CFG দিয়ে modification করতে হবে, যেখানে Parse tree তৈরি & CFG define করতে হবে।

tools :- Bison ← Parser, Semantic Analyzer, CFG

Regular Expression :-

This notation combination of string symbols from alphabet Σ, parenthesis & the operators +, • and *. Here,

+ = union /OR
• = concatenation [AND]
* = star closure
→ সব alphabet এর combination

$(a.b)^* = ab + ba$
b এর পর b হবে না

Language = combination of string. (L).

Certain point on rules need to remember for regular expression:

i) Any termind symbols, i.e, symbols $\in \Sigma$ including $\wedge$ (empty) and $\emptyset$ (null) are regular expression.

ii) The union of the two regular expression is also a regular expression. For example,

$$R_1, R_2 \rightarrow (R_1 + R_2)$$

iii) The concatenation of two regular expression is also a regular expression. For example,

$$R_1, R_2 \rightarrow (R_1 . R_2) \text{ or } R_1 R_2$$

iv) The iteration or star ~~clo~~ closure of a RE is a RE.

$$R \rightarrow R^*, R^+$$

For example, $a^* = \Lambda, a, aa, aaa, aaaa, \ldots$ (empty/null)

$a^+ = a, aa, aaa, aaaa, \ldots$

v) The Regular Expression over Σ are precisely ~~this~~ those obtained recursively by the application of the above rules once or several times.

$(a+b)$, $a \cdot b = ab$.

Write a regular expression for the language L that accepts all the strings have length exactly 2. consider the alphabet, $\Sigma = \{a, b\}$ [Length -> 2^n]

Ans:- $\Sigma = \{a, b\}$

$L = \{aa + ab + ba + bb\}$

$R = aa + ab + ba + bb = a(a+b) + b(a+b)$

$= (a+b)(a+b)$

at least 2 or at most 2 :-

Ans: at least:- $\Sigma = \{a, b\}$

$R = (a+b)(a+b)(a+b)^*$

at most:- $\Sigma = \{a, b\}$

$R = (a+b+\epsilon)(a+b+\epsilon)$

Write a RE for the language, L that accepts all the strings have even numbers of 'a'.

consider the alphabet. $\Sigma = \{a, b\}$

Ans: $\Sigma = \{a, b\}$

$R = b^* (ab^* ab^*)^*$

→ Gall Vidy alay
→ geeks for geeks
→ Tutorial point
→ Java T point

even nb of 'b's

$a^* (ba^* ba^*)^*$

CC (7-8) 15/2/23

token → Syntax Analyzer / Parser → Parse tree

Context Free Grammar

A context free grammar consists of following components

→ Terminal or a set of terminal symbols:

- characters appear in the string generated by the grammar.
- terminals are first components of the tokens output by the lexical analyzer.
- terminal symbols are never appear in left hand side of the production rule & are also on the right hand side.

→ A set of non-terminal symbols.

- which are the place holders for patterns of the terminal symbols that can be generated by the non-terminal symbols.
- the set of strings denoted by non-terminals help define the language generated by the grammar.
- It always appear on the left hand side of the production rule.

→ A set of production rules:

The production of a grammar specify the manner in which the terminals & non-terminals can be combined to form strings. Each production consists of:

- A non-terminal called the head or left side of the production.
- The symbol '→', sometimes '::=' has been used in place of arrow.

→ A body or right side consisting of zero or more terminals and non-terminals. The components of the body ~~desem~~ describe one way in which strings of the non-terminals at the head can be constructed.

Therefore, a context free Grammar can be represented by the notation of - $G = (V, t, S, P)$

$\left[\frac{LHS}{N.T} \rightarrow \frac{R.H.S}{T}\right.$

V = a set of non-terminal symbols (Pα)

t = " " " terminal " (Pα)

S = " " " starting "

P = Production or set of rules (strings)

* terminal এর মাধ্যমে grammer infinite হয় না,

L.H.S R.H.S

$E \rightarrow E+E$

or $\begin{cases} \mid E*E \\ \mid id \end{cases}$

$V = \{E\}$
$T = \{+, *, id\}$
$S = \{E\}$
$P = \{E \rightarrow E+E \mid E*E \mid id\}$

* Let consider the following grammer & identify the all propriety of it:

$E \rightarrow E+T$
$\mid T$
$T \rightarrow F$
$F \rightarrow id$

$V = \{E, T, F\}$
$T = \{+, id\}$
$S = \{E\}$
$P = \{E \rightarrow E+T \mid T,\ T \rightarrow F,\ F \rightarrow id\}$

* How to implement a context free grammar:

→ We need
- a grammar &
- a language or set of strings.

For example, we have a grammar, say

$E \rightarrow E+E$
$\mid E*E$
$\mid id$

and the language, $L = \{id+id*id\}$

Derivation:

For a general defination of derivation consider a non-terminal A in the middle of a sequence of grammar symbols. Suppose $A \to \gamma$ is a production. Then we write

grammar

$\alpha A \beta \Rightarrow \alpha \gamma \beta$

derivative

The symbol $\Rightarrow$ means derives in one step.

For example: $E \to -E$

$| (-E)$

$| id$ → numeric

we have a string $-(id)$.

$E \Rightarrow -E \Rightarrow -(-E) \Rightarrow -(-id)$

* Derivation is 2 types:

→ Left most Derivation (LMD)

→ Right " " (RMD)

CC (7-9) 16/12/23

1) Left Most Derivation (LMD):

In ~~EDT~~ LMDs, the left most non-terminal in each sentential is always chosen. If $\alpha \Rightarrow \beta$ is a step in which the left most non-terminal in α is replaced, we ~~write~~ write $\alpha \underset{lm}{\Rightarrow} \beta$.

Lets consider the following grammar:-

$$E \rightarrow -E \mid (E+E) \mid id$$

We have string on $\Sigma = \{-(-(id+id)+id)\}$

$$E \underset{lm}{\Rightarrow} -E \underset{lm}{\Rightarrow} -(E+E)$$

derive

$$\underset{lm}{\Rightarrow} -(-E+E) \underset{lm}{\Rightarrow} -(-(E+E)+E)$$

$$\underset{lm}{\Rightarrow} -(-(id+E)+E)$$

$$\underset{lm}{\Rightarrow} -(-(id+id)+E) \underset{lm}{\Rightarrow} -(-(id+id)+id)$$

Right Most Derivation (RMD):-

In right most derivations, the right most non-terminal in each sentential is always chosen.

We write $\alpha \underset{rm}{\Rightarrow} \beta$

lets consider the following grammar:

$$E \rightarrow -E \mid (E+E) \mid id$$

We have a string or $L = \{-(-(id+id)+id)\}$

$E \underset{rm}{\Rightarrow} -\underline{E} \underset{rm}{\Rightarrow} -(E+\underline{E})$

$\underset{rm}{\Rightarrow} -(\underline{E}+id)$

$\underset{rm}{\Rightarrow} -(-\underline{E}+id)$

$\underset{rm}{\Rightarrow} -(-(E+\underline{E})+id)$

$\underset{rm}{\Rightarrow} -(-(\underline{E}+id)+id)$

$\underset{rm}{\Rightarrow} -(-(id+id)+id)$

$E \rightarrow c$
$\mid E+E$
$\mid)$
$\mid E*E$
$\mid id$

String: $(id+(id*id))$

LMD:

$E \underset{lm}{\Rightarrow} c$

$\Rightarrow$

23/2/23

CC (T-10)

① "Context Free Grammar is deterministic". Justify the statement applying your analysis.

② Consider the following context free grammar & the string is "abab"

$S \rightarrow aSbS$
$\mid bSaS$
$\mid \epsilon$

i) Give a leftmost derivation of the string &
ii) Give a rightmost

①

2) i) Leftmost:

$S \underset{lm}{\Rightarrow} a\underline{S}bS$

$\underset{lm}{\Rightarrow} a\epsilon bS$

$\underset{lm}{\Rightarrow} ab\underline{S}$

$\underset{lm}{\Rightarrow} ab\underline{a}Sbs$

$\underset{lm}{\Rightarrow} aba\epsilon bs$

$\underset{lm}{\Rightarrow} abab\underline{S}$

$\underset{lm}{\Rightarrow} abab\epsilon$

$\underset{lm}{\Rightarrow} abab$

ii) rightmost:

$S \underset{rm}{\Rightarrow} asb\underline{S}$

$\underset{rm}{\Rightarrow} asb\epsilon$

$\underset{rm}{\Rightarrow} a\underline{S}b$

$\underset{rm}{\Rightarrow} absa\underline{S}b$

$\underset{rm}{\Rightarrow} absa\epsilon b$

$\underset{rm}{\Rightarrow} ab\underline{S}ab$

$\underset{rm}{\Rightarrow} ab\epsilon ab$

$\underset{rm}{\Rightarrow} abab$

$E \rightarrow E+T$
$\quad | E-T$
$\quad | T$

$T \rightarrow T*F$
$\quad | T/F$
$\quad | F$

$F \rightarrow (E)$
$\quad | id$

String:-

$id+(id*(id-id))/id$

is divisible by the grammar

Leftmost:-

$E \Rightarrow \underline{E}+T$

$\Rightarrow \underline{T}+T$

$\Rightarrow \underline{F}+T$

$\Rightarrow id+\underline{T}$

$\Rightarrow id+\underline{T}*F$

$\Rightarrow id+\underline{F}*F$

~~$\Rightarrow id+(\underline{E})*F$~~

~~$\Rightarrow id+(id*(\underline{E})$~~

~~$\Rightarrow id+(id*(E-T)$~~

$\Rightarrow id+(\underline{E})*F$

$\Rightarrow id+(\underline{T})*F$

$\Rightarrow id+(\underline{T}*F)*F$

$\Rightarrow id+(F*F)*F$

$\Rightarrow id+(id*F)*F$

$\Rightarrow id+(id*(E)*F$

$\Rightarrow id+(id*(E-T)*F$

$\Rightarrow id+(id*(T-T)*F$

$\Rightarrow id+(id*(F-T)*F$

$\Rightarrow id+(id*(id-T)*F$

$\Rightarrow id+(id*(id-F)*F$

$\Rightarrow id+(id*(id-id)*F$

$\Rightarrow id+(id*(id-id)*$